Assignment 4: Spell Checker

In this assignment, you will practice reading and writing from files using Java (aka file I/O) and manipulating strings, skills that form the foundation of working with databases, data science, big data, and more.

The activities in this week's Recitation will help you practice for this assignment, so it is highly recommended that you attend the live sessions or review the recordings and be sure to attempt the recitation activities.

Copying and pasting from assignment PDFs will cause issues when running your code. It is best practice to type your code out.

Goal

In this assignment, you will be building a simple spell checker. The spell checker that you build will perform three tasks:

1. Spot the misspelled words in a file by checking each word in the file against a provided dictionary.
2. Provide the user with a list of alternative words to replace any misspelled word.
3. Write a new file with the corrected words as selected by frr the user. Please note that it would take some work to maintain the original file's punctuation. Do not worry about that for this assignment.

Specifications

A list of correctly spelled English words is provided as a reference in the file called `engDictionary.txt`, one word per line of the file. The user file to be checked is a text file where each space-separated word of the file is to be examined.

You can safely assume that every word in the user file is lowercase. You can also assume that all punctuation has been removed. So your file looks like this for example (taken from the first chapter of "The Adventures of Tom Sawyer" by "Mark Twain")

the old lady pulled her spectacles down and looked over them about the room then she put them up and looked out under them she seldom or never looked through them for so small a thing as a boy they were her state pair the pride of her heart and were built for style not service she could have seen through a pair of stove lids just as well she looked perplexed for a moment and then said not fiercely but still loud enough for the furniture to hear

We are going to ignore punctuation completely (i.e., input files have no punctuation), so this means no periods, question marks, quotations, apostrophes-- just think of this as a stream of words--line breaks are allowed.
The final spell-checked file that is produced will not have the same case specifications. That's ok! That file might also not retain the line breaks. That's ok, too! If you want to add logic that deals with these in a special manner, you may, just make sure that it does not break the specifications outlined below.

If a word from the file does not exist in the provided list, then it is assumed to be misspelled and a set of alternatives provided to the user. The user has three options for the provided candidate list:

- The user types the letter 'r', indicating that they want to replace the word with one of the words provided as suggested spellings. They then select, by number, one of the candidates. The word in the file is replaced in the output file by the selected number.
- Indicate that they wish to leave the word as is. This is done by the user just typing the letter 'a'.
- Indicate that they wish to provide the alternative by typing it in directly. The program replaces the word in the file with the user-provided replacement in the output file. The user does this by first entering the letter 't'. They then proceed to type the word.

Example User Interaction

Typical interaction will be something like the following. The blue text indicates what is actually displayed to the user. The normal (black) text is just for explanation purposes

The word 'morbit' is misspelled.
The following suggestions are available
1. 'morbid'
2. 'orbit'.
Press 'r' for replace, 'a' for accept as is, 't' for type in manually.
User presses 'r'
Your word will now be replaced with one of the suggestions
Enter the number corresponding to the word that you want to use for replacement.
User enters 1
In the output file, morbit will be replaced with morbid.

The word 'automagically' is misspelled.
The following suggestions are available
1. automatically.

Press ‘r’ for replace, ‘a’ for accept as is, ‘t’ for type in manually.

User presses ‘a’ and in the output file, ‘automagically’ stays the same.

The word ‘ewook’ is misspelled. The following suggestions are available
1. wok
2. woo
3. awoke
Press ‘r’ for replace, ‘a’ for accept as is, ‘t’ for type in manually.

User presses ‘t’
Please type the word that will be used as the replacement in the output file.
User types ’ewok’
In the output file, this word is changed to ewok.
The rare case is when the spell checker cannot come up with any suggestion at all. Note that this case is slightly different so be aware of it when you program.

The word ‘sleepyyyyyyyy’ is misspelled.
There are 0 suggestions in our dictionary for this word.
Press ‘a’ for accept as is, ‘t’ for type in manually.
User presses ‘t’
Please type the word that will be used as the replacement in the output file.
User types ‘sleepy’

If the user is trying to be silly, and enters something other than ‘r’, ‘a’ or ‘t’ and/or a number that is outside of the range of options when they have decided to do a replace, please tell the user politely that they have to try again. Do not let the program exit or crash in a user-unfriendly manner.

Breaking Down the Assignment

We would like you to break down this assignment into the following pieces.

1. Prompt the user for the name of a file to spell check. The program will spell check each word and then write a new file with the name of the original file plus the suffix ‘_chk’. Thus if the file being checked is shopping.txt, the spell checked output will be shopping_chk.txt. Note that the file suffix is preserved!

2. Create a method that checks a word of the user file against words from the dictionary file and see if that word is there (more details below in the functions list).
3. Write a loop which reads from the input file and goes through it a word at a time. Each word of the file is checked to see if it is spelled correctly based on the reference list. If it is spelled correctly, just write it directly to the output file. If it is not spelled correctly, then provide the user with the options discussed above. Depending upon their responses, write the corresponding word to the output file.
4. Output the name of the updated spell-checked file. The user can then open up that file via Finder/Windows Explorer (your program does not need to do this) and see the result of the spell checking.

Breaking Down the Assignment Even Further
(Methods that need to be written)

In order to help you out, here is a class called `WordRecommender` along with some methods that we want you to write. Create this class.

Remember that we still would like you to follow good design principles. So **think about other classes that might be useful.** Combine them in a manner that makes the most sense to solve this homework.

We will first have you make a class called `WordRecommender` which has at least one instance variable – a `String filename` that has a dictionary of words. For your assignment this will be equal to `engDictionary.txt`, but we want to ensure that the spellchecker can run with any dictionary. Hence the flexibility. This should be set by the constructor, which takes one argument:

```
public WordRecommender(String fileName)
```

When you call this constructor elsewhere in your program, you may hard-code it to be `"engDictionary.txt"` (you do not need to ask for user input).

The `WordRecommender` Class will have the following methods in it:

1. `public double getSimilarityMetric(String word1, String word2)`

given two words, this function computes two measures of similarity and returns the average.

leftSimilarity (a made-up metric) – the number of letters that match up between word1 and word2 as we go from left to right and compare character by character.
So the leftSimilarity for 'oblige' and 'oblivion' is 4, the leftSimilarity for 'aghast' and 'gross' is 1.

For the "oblige" and "oblivion" example the first character is an o for both strings, the second character is a b for both, the third character is an l, the fourth character is an i and then nothing else lines up.

rightSimilarity (another made-up metric) – the number of letters that match up, but this time going from right to left.

So the rightSimilarity for 'oblige' and 'oblivion' is 1.
the rightSimilarity for 'aghast' and 'gross' is 2.

For the aghast and gross example, the last character is a t and an s respectively so those do not count, the second last character is an s in both cases, the a and the o do not line up, the h and the r do not line up, and finally the fifth character from the end is a g in both cases and therefore that contributes to the score as well.

Finally to get the similarity score, take the average of leftSimilarity and rightSimilarity and return that value. So getSimilarityMetric('oblige', 'oblivion') will return (4+1)/2.0 = 2.5

`2. public ArrayList<String> getWordSuggestions(String word, int n, double commonPercent, int topN)` –
given an incorrect word, return a list of legal word suggestions as per an algorithm given below. You can safely assume this function will only be called with a word that is not already present in the dictionary. `commonPercent` should be a double between 0.0, corresponding to 0%, and 1.0, corresponding to 100%.

To come up with a list of candidate words, we first come up with a list of candidate words that satisfy both of these two criteria:

a) candidate word length is `word` length +/- `n` characters .
b) have at least commonPercent% of the letters in common.

We will define commonPercent mathematically as follows: Consider two words w1 and w2. Create the set of letters in w1, call that S1 (remember it is a set so repeated letters show up only once). Create the set of letters in w2, call that S2. Then commonPercent is defined as

Number of elements in S1 intersected with S2 / Number of elements in S1 union S2.

Or

Number of letters that are common across the two sets (Set 1 and Set 2) divided by the Total number of letters in Set 1 and Set 2 (with each letter counting exactly once).

Here are two examples:

w1 - committee. Then S1 = {c, o, m, i, t, e}
w2 - comet. Then S2 = {c , o, m, e, t}
Then the numerator is the letters in common {c, o, m, e, t}. 5 of them
The denominator is all the letters {c, o, m, e, t, i}. 6 of them

Therefore the percent is 5/6

w1 - gardener - {g, a, r, d, e, n}
w2 - nerdier - {n, e, r, d, i}
Numerator = letters in common = {n, e, r, d}. 4 of them
Denominator = all the letters = {g, a, r, d, e, n, i}. 7 of them
Common percent = 4/7

Next, for all the words that satisfy these two criteria, use the similarity metric (see above) and return the topN number of them. If there are fewer than topN words that satisfy the criteria, just return all of them.

You are not allowed to use the Collections.sort method from Java in this part. Usage of that method will result in the loss of 4 points.

This method involves more work so please ensure that you have written the previous methods first.

```
3. public ArrayList<String> getWordsWithCommonLetters(String word,
ArrayList<String> listOfWords, int n)
```

This method is to give you more practice with string manipulation-- it won't be used in the Spell Checker.

Given a word and a list of words from a dictionary, return the list of words in the dictionary that have at least (>=) n letters in common. For the purposes of this method, we will only consider the distinct letters in the word. The position of the letters doesn't matter.

Consider a wordList to contain ['ban', 'bang', 'mange', 'gang', 'cling', 'loo'] and the word we have is 'cloong'.

Then we want the result of
`getWordsWithCommonLetters('cloong', wordList, 2)` will return `['bang', 'mange', 'gang', 'cling', 'loo']`
and
`getWordsWithCommonLetters('cloong', wordList, 3)` will return `['cling'].`

Note that we only count distinct letters, which is why the word 'loo' does not appear in the second example.

4. `public String prettyPrint(ArrayList<String> list)`

Finally, here is a method that you need to write purely for display purposes. This method takes an ArrayList and returns a String which when printed will have the list elements with a number in front of them

`prettyPrint(["biker", "tiger", "bigger"])` returns the string "1. biker\n2. tiger\n3. bigger\n" so that when you print it you get

1. biker
2. tiger
3. bigger

Evaluate the design

In a plain text file named README, you should assess the design, and provide one recommendation for how it could be made better. Think about the principles we've discussed in class such as cohesion, coupling, and DRY, among other considerations.

Please note that you have to assess the design and not the algorithms. So do not provide us with new methods for computing word similarities. We are looking for design changes. If you are stuck, look at the definition of the keywords like Cohesion, Coupling, DRY etc. We just need one suggested change.

What to Submit

Please submit the following files

README - this is a plain text file which explains the design change you would like to make.
WordRecommender.java
engDictionary.txt
and
Any other java files that you made in the process of creating your spell checker.*

*There will be other files that you will create (hint hint). **Make sure that you submit them as well.** They will count as part of your grade for the assignment. Remember that if your code

does not compile, we will not be able to give you a score. Also those extra files might amount to your design being even better. (Remember, one of the learning goals for this assignment is design!)

Suggested approach

Start by writing and unit testing WordRecommender. Since the method stubs in this class have already been provided to you, and there are validation tests on Codio, you can get quick feedback. *Please note that as with other assignments passing the validation test does not mean that you will receive full credit. Not all tests are revealed to you.*

Now consider how you would want to program the interaction with the user. How would you read from the file? How would you write to the corrected file?

Testing this code is not that easy when you stick to the engDictionary.txt file that we provided. Remember that your code should work for any text file that has a list of accepted words. Our recommendation would therefore be to test with a list of about 10-20 legitimate words and having a short passage that has spelling errors.